Eetu Röpelinen

Pelinkehitykseen tarkoitetun verkkokirjaston suunnittelu ja toteutus

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Tradenomi
Tietojenkäsittelyn koulutusohjelma
Syksy 2013

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Tradenomi	Koulutusohjelma Tietojenkäsittelyn koulutusohjelma
Tekijä(t) Eetu Röpelinen	
Työn nimi Pelinkehitykseen tarkoitetun verkkokirjaston suunnittelu ja toteutus	
Vaihtoehtoiset ammattiopinnot Peliohjelmointi	Ohjaaja(t) Mikko Romppainen
	Toimeksiantaja -
Aika Syksy 2013	Sivumäärä ja liitteet 33

Opinnäytetyön tarkoituksena on tutkia ja selvittää mitä protokollia pelinkehitykseen soveltuvan verkkokirjaston tulisi käyttää ja mitä ominaisuuksia siitä tulisi löytyä. Aihe on ajankohtainen, koska nykyisin lähes kaikista videopeleistä löytyy jonkinlainen moninpeli- tai verkkokomponentti. Verkkokirjasto toimiikin tällaisten komponenttejen pohjana.

Aluksi opinnäytteessä tutustutaan TCP/IP-protokollaperheeseen ja etenkin sen TCP- ja UDP-verkkoprotokolliin. Tämän jälkeen tutkitaan UDP-verkkoprotokollan laajentamista pelinkehitykseen paremmin soveltuvaksi. Teoriaosiossa tutustutaan myös Windows-käyttöjärjestelmille kehitettyyn Windows Sockets-ohjelmointirajapintaan. Lopuksi tutustutaan vielä prosessien välisiin funktiokutsuihin ja vertaillaan RPC-protokollaa ja CORBA-arkkitehtuuria toisiinsa.

Käytännön työn tuloksena syntyi pelinkehitykseen soveltuva verkkokirjasto. Verkkokirjasto mahdollistaa yhteydellisen tiedonsiirron UPD-verkkoprotokollaa käyttäen, sekä prosessien välisten funtiokutsujen suorittamisen. Verkkokirjasto saavutti kaikki sille asetetut tavoitteet eikä sen toiminnassa havaittu ongelmia. Opinnäytetyö onkin onnistunut ohjelmointiprojektin sekä tekijän oppimisen näkökulmasta.

Kieli	Suomi
Asiasanat	Ohjelmointi, Verkkokirjasto, Peli
Säilytyspaikka	☒ Verkkokirjasto Theseus ☒ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS
ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Eetu Röpelinen	
Title Design and Implementation of a Network Library Meant for Game Development Use	
Optional Professional Studies Game Programming	Instructor(s) Mikko Romppainen
	Commissioned by -
Date Fall 2013	Total Number of Pages and Appendices 33

The purpose of this thesis is to study and investigate which protocols a network library meant for game development use should use and what features it should have. The topic is current because nowadays almost all video games have some kind of a multiplayer or network component. A network library works as a foundation for these kinds of components.

At first, there is an introduction to the TCP/IP protocol suite and especially to its TCP and UDP protocols. After this, techniques of improving UPD to better suit game development are studied. The theory part of this thesis also takes a look at the Windows Sockets API for Windows operating systems. Lastly, this thesis studies remote procedure calls and compares the RPC protocol with the CORBA architecture.

As a part of this thesis a network library meant for game development use was created. The network library allows connection oriented data transfer using UDP and executing remote procedure calls. The network library achieved all the goals set for it and no flaws were detected in its functionality. Consequently, this thesis is a success as a programming project and a learning experience.

Language of Thesis	Finnish
Keywords	Programming, Network Library, Game
Deposited at	☒ Electronic library Theseus ☒ Library of Kajaani University of Applied Sciences

ALKUSANAT

Innoituksen tähän opinnäytetyöhön sain koulun verkkopeliohjelmoinnin kurssin aikana. Huomasin olevani kiinnostunut verkko-ohjelmoinnista yleensä ja näin verkkokoodissa olevien virheiden etsinnän ja korjaamisen erittäin tyydyttävänä haasteena. Aiheen rajaus yleisestä verkko-ohjelmoinnista tähän kyseiseen tarkemmin rajattuun aiheeseen oli kuitenkin aikaavievä ja haasteellinen prosessi. Lopullisen aiheen raajaamiseen sainkin korvaamatonta apua tämän opinnäytetyön ohjaajalta Mikko Romppaiselta.

Tämän opinnäytetyön tavoitteena on verkkokirjaston tuottamisen lisäksi oppia verkko-ohjelmoinnista mahdollisimman paljon sekä kehittyä ohjelmoijana mahdollisimman monella osa-alueella. Tarkoituksena on tutustua etenkin matalan tason verkko-ohjelmointiin sekä etenkin pelien verkkokirjastoille asettamiin vaatimuksiin. Koska opinnäytetyöllä ei ole toimeksiantajaa, on tavoitteena tuottaa verkkokirjasto, jota voin itse käyttää portfolio materiaalina ja jonka toivottavasti pystyn julkaisemaan myös avoimen lähdekoodin ohjelmistona muiden käyttöön.

SISÄLLYS

SYMBOLILUETTELO

API	Ohjelmointirajapinta (engl. Application Programming Interface) määrittelee kuinka ohjelmistokomponentit vaihtavat tietoa keskenään.
CORBA	CORBA (engl. Common Object Request Broker Architecture) mahdollistaa eri ohjelmointikielillä valmistettujen ja eri järjestelmillä pyörivien ohjelmien yhteistoiminnan.
Debuggeri	Debuggeri on ohjelmisto jota käytetään ohjelmointivirheiden etsimiseen ja korjaamiseen.
Intelli-sense	Lyhenne sanoista intelligent code sense. Intelli-sense on osa ohjelmointiympäristöä ja se ilmoittaa lähes reaaliajassa ohjelmoijan tekemistä syntaksivirheistä.
IP	TCP/IP:n verkkokerroksella sijaitseva protokolla, joka mahdollistaa pakettien reitittämisen lähi- ja laajaverkossa.
Jitter	Jitter on latenssissa esiintyvää vaihtelua lyhyellä aikavälillä. Tämä tarkoittaa esimerkiksi peräkkäisten pakettien latenssien eroja.
Latenssi	Latenssi tarkoittaa järjestelmässä esiintyvää viivettä. Tämän opinnäytetyön tapauksessa latenssi tarkoittaa aikaa, joka verkossa kulkevalta paketilta kuluu matkata vastaanottajalle.
OSI-malli	Seitsenkerroksinen käsitemalli pakettivälitteisestä tietoliikenteestä.
RFC	RFC (engl. Remote Function Call), eli prosessien välinen funktiokutsu, on prosessien välistä kommunikaatiota, joka mahdollistaa funktion kutsumisen toisesta prosessista käsin.
RPC	RPC (engl. Remote Procedure Call) on prosessien välistä kommunikaatiota, joka mahdollistaa funktioiden kutsumisen toisesta ohjelmistosta. Ohjelmistot ovat yleensä eri tietokoneilla ja kutsu suoritetaan verkon välityksellä.

RTT	RTT (engl. Round Trip Time) tarkoittaa aikaa joka verkossa kulkevalta paketilta kuluu matkata vastaanottajalle ja takaisin. RTT ei välttämättä ole tuplasti latenssi.
TCP	TCP/IP:n kuljetuskerroksen yhteydellinen ja luotettava tietoliikenneprotokolla.
TCP/IP	Internetin perustana toimiva protokollaperhe.
UDP	TCP/IP:n kuljetuskerroksen yhteydetön tietoliikenneprotokolla.

1 JOHDANTO

Opinnäytetyön aiheena on tutkia verkkokirjaston toteutukseen vaadittavia ohjelmointitekniikoita sekä pelien verkkokirjastoille asettamia vaatimuksia. Tutkimuksen aikana kerätyn tiedon pohjalta on tarkoituksena toteuttaa erityisesti pelien kehittämiseen soveltuva yksinkertainen verkkokirjasto. Opinnäytetyön aihe on ajankohtainen, koska nykyisin lähes kaikissa videopeleissä on jonkinlainen moninpeli- tai verkkokomponentti. Verkkokirjasto toimiikin kyseisten komponenttien pohjana.

Opinnäytetyössä ei ole tarkoituksena tutustua verkkopelien pelimekaniikan toimintaan, vaan opinnäytetyössä keskitytään nimenomaan matalan tason verkko-ohjelmointiin. Tärkeimpiä asioita ovat oikean tiedonsiirtoprotokollan valinta sekä luotettavan ja yhteydellisen tiedonsiirron tarpeellisuus verkkopeleissä. Yksinkertaisten ja lähes pakollisten ominaisuuksien lisäksi opinnäytetyössä tullaan tutustumaan prosessien välisten funktiokutsujen toimintaan ja toteuttamiseen. Prosessien väliset funktiokutsut valittiin opinnäytetyöhön, koska niistä voi olla suurtakin hyötyä videopelin kehityksen aikana, vaikka ominaisuutta ei lopullisessa pelissä käytettäisikään. Ominaisuus lisää myös käytännön työn haastavuutta.

Käytännön osuudessa tarkoituksena on toteuttaa yksinkertainen verkkokirjasto. Tulen käyttämään teoriaosuuden aikana keräämääni tietoa verkkokirjaston suunnittelun pohjana. Tulen myös toteuttamaan hyvin yksinkertaisen prototyyppi pelin, jolla testaan verkkokirjastoa. Peli tulee sisältämään vain minimimäärän pelillisiä ominaisuuksia, koska pelin tarkoitus on demonstroida visuaalisesti verkkokirjaston toimintaa sekä testata verkkokirjaston käytettävyyttä.

2 TEORIA

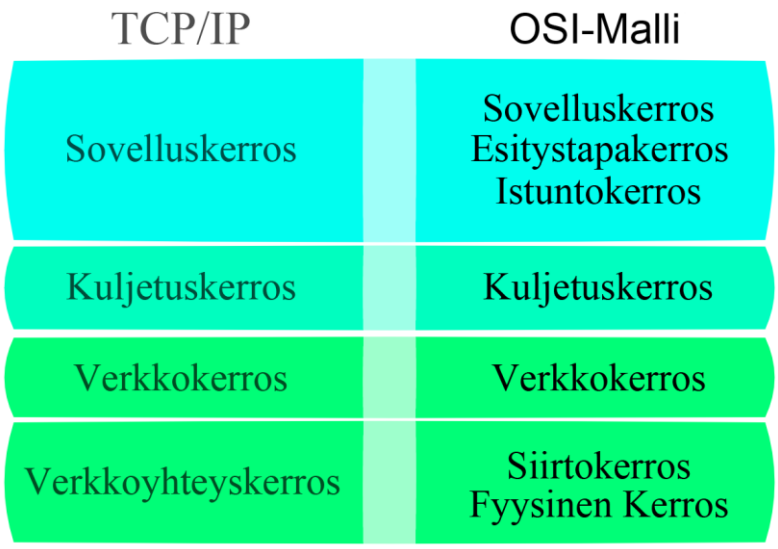
Teoriaosio avaa lukijalle verkkokirjastoprojektissa käytettyjen protokollien sekä järjestelmien toimintaa ja tarkoitusta. Teoriaosiossa ei keskitytä pelkästään niihin protokolliin sekä järjestelmiin joita projektissa loppujen lopuksi tullaan käyttämään, vaan siinä käsitellään myös vaihtoehtoisia tekniikoita. Teoriaosio myös perustelee miksi tietyt protokollat sekä ominaisuudet ovat tai eivät ole käyttökelpoisia pelinkehitykseen tarkoitetun verkkokirjaston toteutuksessa.

2.1 TCP/IP-protokollaperhe

TCP/IP-protokollaperhe toimii internetin perustana ja nykyisin lähes kaikkia sen protokollia kehittää Internet Engineering Task Force (IETF). IETF:n toimintaa ohjaa itsehallinnollinen organisaatio Internet Architecture Board (IAB). TCP/IP-protokolla on hyvin dokumentoitu ja dokumentit ovat vapaasti saatavilla IETF:n sivuilta. (Javvin 2007, 8)

Vaikka ei olekkaan olemassa yhtä virallista tapaa esittää TCP/IP-protokollaperheen arkkitehtuuria kerroksittaisena mallina, TCP/IP:n arkkitehtuuria voidaan kuitenkin verrata seitsenkerroksiseen OSI-malliin. Useimmiten TCP/IP:n arkkitehtuurin kerroksittaisessa esitystavassa on kolmesta viiteen kerrosta. (Javvin 2007, 4)

Esimerkiksi kaikki OSI-mallissa kuljetuskerroksen yläpuolella olevat kerrokset voidaan yksinkertaistaa TCP/IP:n kerroksittaisessa mallissa sovelluskerrokseksi. Vastaavasti kaikki OSI-mallin verkkokerroksen alapuolella olevat kerrokset voidaan yksinkertaistaa verkkoyhteyskerrokseksi. Tällöin TCP/IP:n arkkitehtuuri esitetään neljässä kerroksessa, jotka ovat ylhäältä alaspäin lueteltuna sovellus-, kuljetus-, verkko- sekä verkkoyhteyskerros. Kyseinen esitystapa on nähtävissä kuviossa 1. Tulen tässä opinnäytteessä käyttämään TCP/IP:n kerroksien kuvaukseen edellä mainittua neljäkerroksista mallia. (Javvin 2007, 4)



Kuvio 1. TCP/IP:n neljä kerroksinen esitysmalli verrattuna OSI-malliin.

Kun dataa siirretään sovellukselta fyysiselle verkkokerrokselle, jokainen kerros käsittelee kaikkea ylemmältä kerrokselta saamaansa informaatiota datana. Tähän dataan kerros lisää oman otsikkotietonsa ja sen jälkeen lähettää sen alemmalle kerrokselle. Kun tietoa vastaanotetaan fyysiseltä verkkokerrokselta sovellukselle, jokainen kerros vuorollaan poistaa oman otsikkotietonsa datasta ja lähettää datan ylemmälle kerrokselle käsiteltäväksi. (Javvin 2007, 4)

TCP/IP-protokollaperheen protokollista käsittelen tässä opinnäytetyössä verkkokerroksella sijaitsevaa IP:tä (engl. Internet Protocol), sekä kuljetuskerroksella sijaitsevaa TCP:tä (engl. Transmission Control Protocol) ja UDP:tä (engl. User Datagram Protocol). TCP ja UDP ovat TCP/IP-protokollaperheen ainoat kuljetuskerroksella sijaitsevat protokollat ja IP on TCP/IP:n ensisijainen protokolla verkkokerroksella. (Javvin 2007, 4)

2.1.1 IP

IP:stä (engl. Internet Protocol) on käytössä kaksi versiota, IPv4 ja IPv6. Yleensä IP:stä puhuttaessa viitataan vanhempaan IPv4:ään ja IP:n uudemmasta versiosta puhuttaessa käytetään versionumeron sisältävää lyhennettä. Myös tässä opinnäytteessä IP:stä puhuttaessa viittaan IPv4:ään. Vaikka IPv6 onkin selkeä kehitysaskel IPv4:ään verrattuna, tullaan IPv4:ää käyttämään vielä kauan IPv6:en rinnalla. Tämä johtuu IPv4:ää käyttävien laitteiden valtavasta

määrästä ja siitä, että niiden korvaaminen IPv6:tta tukevilla laitteilla on aikaa vievää. (Javvin 2007, 54-56)

IP sijaitsee TCP/IP:n verkkokerroksella (OSI-mallin kolmas kerros) ja se mahdollistaa pakettien reitittämisen niin lähiverkossa kuin laajaverkossakin. IP:n tärkeimmät tehtävät ovat yhteydetön pakettien toimittaminen verkossa parhaan yrityksen toimituksella (engl. best-effort delivery) sekä pakettien paloittelu ja uudelleen kokoaminen. Parhaan yrityksen toimitus tarkoittaa, että pakettien matka-ajasta verkossa sekä pakettien perille saapumisesta ei anneta takuita (Armitage & Claypool & Branch 2006, 42). Pakettien paloittelu ja uudelleen kokoaminen taas mahdollistaa tiedon siirron eri kokoisia paketteja tukevilla datayhteyksillä. (Javvin 2007, 54)

IP-verkossa olevat laitteet tunnistetaan toisistaan IP-osoitteen avulla ja IP-osoite onkin välttämätön, jotta paketit saadaan toimitettua verkossa oikealle vastaanottajalle. Jokaisella IP-verkossa olevalla laitteella on uniikki IP-osoite. IP-osoite IPv4-verkossa on 32-bittinen ja IPv6-verkossa se on 128-bittinen. Tämä onkin IP:n neljännen ja kuudennen version näkyvin ero ja IP-osoitteen kasvattaminen 128-bittiseksi mahdollistaa moninkertaisen laitemäärän internetissä. (Javvin 2007, 54-55)

Kun tietoa lähetetään ja vastaanotetaan IP:n välityksellä, lähetettävä data pilkotaan paketeiksi. Jokaiseen pakettiin sisällytetään sekä lähettäjän että vastaanottajan IP-osoite. Koska data on jaettu pienemmiksi paketeiksi, on mahdollista lähettää data vastaanottajalleen useita reittejä pitkin. Nämä paketit voivat saapua vastaanottajalleen eri järjestyksessä kuin ne on lähetetty ja osa paketeista saattaa kadota matkalle. IP:stä riippumattomat protokollat, kuten TCP, huolehtivat pakettien järjestämisestä oikeaan järjestykseen, sekä mahdollisesti hävinneiden pakettien uudelleen lähetyksestä. IP huolehtii vain pakettien toimittamisesta. (Javvin 2007, 54)

2.1.2 TCP

TCP (engl. Transmission Control Protocol) on TCP/IP-protokollaperheen kuljetuskerroksen protokolla ja IP:n kanssa se muodostaa TCP/IP:n ytimen. TCP tarjoaa virtuaalisen yhteyden sekä datan jonotoimituksen (engl. stream delivery). TCP tarjoaa myös luotettavan tiedonsiirron, vuonhallinnan, sekä tuen kaksisuuntaiselle (engl. full-duplex) ja lomitetulle (engl. multiplexing) tiedonsiirrolle. Vaikka TCP esitetäänkin TCP/IP-protokollaperheen osana, on

se myös itsenäinen protokolla, jota voidaan käyttä muidenkin tiedonvälitysjärjestelmien kanssa (Comer 2000, 209). (Javvin 2007, 48)

Koska samalla laitteella voi olla yhtä aikaa käynnissä useita verkko-ohjelmia, TCP:n täytyy pystyä toimittamaan paketti oikealle ohjelmalle. Tämä on mahdollista porttinumeroiden avulla. IP-osoitteen ja porttinumeron yhdistelmää kutsutaan pistokkeeksi. TCP luokin virtuaalisen yhteyden lähettäjän ja vastaanottajan porttien välille. (Javvin 2007, 48)

TCP käyttää neljää ruuhkanhallintamenetelmää. Nämä menetelmät ovat hidas aloitus, ruuhkan välttely, nopea uudelleenlähetys sekä nopea palautuminen. Hitaan aloituksen ja ruuhkanvälttelyn tavoitteena on estää lähettäjää lähettämästä enemmän dataa kuin verkko pystyy siirtämään. Nopean uudelleenlähetyksen ja nopean palautumisen tarkoituksena on havaita ja uudelleenlähettää hävinneet paketit mahdollisimman nopeasti. (Allman & Paxson & Blanton 2009, 1, 4, 8)

Vuonhallinnan seurauksena TCP saattaa käyttää useita sekunteja kadonneiden pakettejen uudelleenlähettämiseen. Koska uusia paketteja ei lähetetä ennen kuin vanhat paketit on saatu toimitettua perille, joutuu uusi data odottamaan vanhan datan perille pääsyä. Tästä syystä TCP ei sovellu käytettäväksi ohjelmistoissa, joissa tietoa on toimitettava reaaliaikaisesti. Tällaisia ohjelmistoja ovat esimerkiksi verkon välityksellä toimivat videopelit. (Armitage & Claypool & Branch 2006, 45)

TCP puskuroi automaattisesti lähetettävää dataa ja jakaa sen paketteihin, jotka kokonsa puolesta käyttävät verkkokaistaa tehokkaasti. Tämä voi olla ongelmana etenkin videopeleissä, joissa datan on kuljettava mahdollisimman nopeasti. Datan puskurointi muodostuu ongelmaksi, jos ohjelman lähettämä data on todella pientä. Tällöin dataa jää puskuriin odottamaan kunnes sitä on kertynyt tarpeeksi lähetettäväksi. Tällainen puskuroinnin aiheuttama viive voi tehdä pelistä pelikelvottoman. (Fiedler b. 2008)

2.1.3 UDP

UDP (engl. User Datagram Protocol) on OSI-mallin neljännen kerroksen yhteydetön tiedonsiirtoprotokolla. UDP on TCP:hen verrattuna erittäin yksinkertainen protokolla, koska UDP ei lisää IP:n päälle luotettavaa tiedonsiirtoa, vuonhallintaa tai virheenkorjaustoimintoja.

UDP onkin lähinnä rajapinta IP:n ja ylemmän kerroksen sovellusten välillä. Yksinkertaisuutensa vuoksi UDP:n otsikko on huomattavasti TCP:n otsikkoa pienempi. Tämän takia UDP käyttää vähemmän verkkokaistaa kuin TCP. UDP onkin erityisen hyödyllinen ohjelmistoissa joissa ohjelmisto itse tarjoaa tarvittavat vuonhallinta ja virheenkorjaus ominaisuudet. (Javvin 2007, 49)

Koska UDP ei tarjoa luotettavaa tiedonsiirtoa, vuonhallintaa tai virheenkorjausta, UDP-paketit voivat hävitä, duplikoitua, saapua väärässä järjestyksessä sekä saapua nopeammin kuin vastaanottaja niitä pystyy käsittelemään. Tästä syystä ohjelmisto joka käyttää UDP:tä ottaa täyden vastuun kaikkien edellämainittujen ominaisuuksien toteutuksesta. (Comer 2000, 198-199)

TCP-liikenteellä on haitallinen vaikutus UDP-liikenteeseen, joka kulkee saman solmun kautta. Tämä solmu toimii yleensä pullonkaulana lähiverkosta laajaverkkoon. Etenkin useat yhtäaikaiset TCP-yhteydet, joiden ruuhkaikkunat ovat synkronisoituneet, ovat erittäin haitallisia UDP-yhteyksille. Ruuhkaikkuna on TCP:n muuttuja, joka rajoittaa lähetettävän datan määrää (Allman & Paxson & Blanton 2009, 3). TCP-liikenne on erityisen haitallista UDP-liikenteelle, jossa pakettikoot ovat suuria ja lähetystaajuus on pieni. (Sawashima & Hori & Sunahara & Oie 1997)

2.2 UDP:n laajentaminen verkkomoninpelejä varten

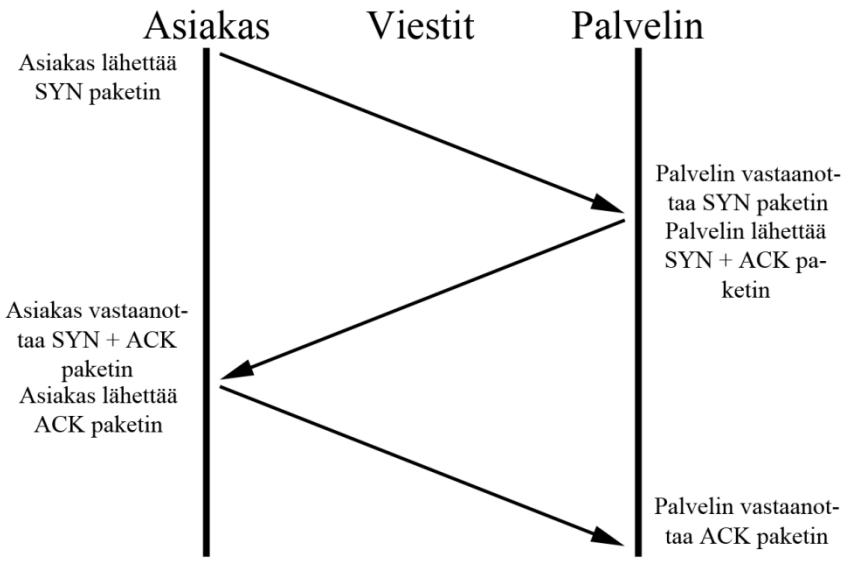
Tämä luku käsittelee yhteydellisen ja luotettavan tiedonsiirron sekä latenssin seurannan toteuttamista UDP:n päälle. Tietyissä tilanteissa yhteydellisestä sekä luotettavasta tiedonsiirrosta on hyötyä videopelejä kehitettäessä. Vaikka TCP:stä nämä ominaisuudet löytyvät, emme halua käyttää sitä luvussa 2.1.2 luetelluista syistä. Tämän takia ainoa vaihtoehto on toteuttaa nämä ominaisuudet itse UDP:n päälle. Tässä luvussa käsitellään tapoja toteuttaa edellämainitut ominaisuudet, ja koska ne on toteutettu TCP:hen käytän TCP:tä esimerkkinä.

2.2.1 Yhteydellinen tiedonsiirto

Kun tietoa siirretään internetissä lähettävän ja vastaanottavan tietokoneen välille ei muodosteta suoraa fyysistä yhteyttä. Sen sijaan IP reitittää lähetetyt datapaketit useiden tietokoneiden

kautta lähettäjältä määränpäähänsä. Tietokoneita joiden kautta paketit kulkevat ei voida tietää etukäteen ja reitti voi muuttua lähetyksen aikana. Tästä syystä yhteydellisestä tiedonsiirrosta puhuttaessa käytetään termiä virtuaalinen yhteys. Virtuaalinen yhteys onkin vain lähettäjän ja vastaanottajan tietoisuus toisistaan. (Fiedler c. 2008)

TCP luo yhteyden kolmitiekättelyn avulla. Ensin asiakas lähettää palvelimelle paketin, jonka otsikkotiedon SYN eli synkronointibitti on asetettu. Palvelin lähettää vastaukseksi paketin, jonka SYN bitti sekä ACK eli kuittausbitti on asetettu. Kättelyn kolmas viesti on palvelimen lähettämä paketti jonka ACK bitti on asetettu. Viimeisen viestin tarkoituksena on ilmoittaa palvelimelle, että molemmat osapuolet ovat yhtä mieltä yhteyden muodostamisen onnistumisesta. Kolmitiekättelyn toiminta on esitetty kuviossa 2. (Comer 2000, 237-238 & Javvin 2007, 48-49)



Kuvio 2. Kolmitiekättelyn toiminta (Mukaillen Comer 2000, 238).

Virtuaalisen yhteyden muodostaminen on mahdollista myös ilman TCP:n käyttämää monivaiheista kolmitiekättelyä. Esimerkiksi jokaiseen lähetettävään pakettiin voidaan liittää otsikko, joka sisältää ohjelmiston tunnistenumeron. Kun palvelin saa ensimmäisen paketin jonka tunnistenumero on oikea, palvelin toteaa yhteyden luoduksi. Asiakas lähettää paketteja olettaen yhteyden muodostuneen. Kun palvelin saa ensimmäisen paketin, se tallentaa IP-osoitteen ja portin josta paketti oli lähetetty ja aloittaa omien pakettiensa lähettämisen tallennettuun osoitteeseen. (Fiedler c. 2008)

2.2.2 Luotettava tiedonsiirto

Moninpelattavat toimintapelit vaativat tasaista pakettivirtaa noin 10-30 paketin sekuntivauhdilla. Useimmiten näiden pakettien sisältämä data on niin lyhytkestoista, että vain uusin data on käyttökelpoista. Tällainen data sisältää esimerkiksi pelaajan syötteet ja jokaisen pelihahmon paikan, rotaation sekä suunnan pelimaailmassa. (Fiedler a. 2008)

Kuten jo aikaisemmin on mainittu, kadottaessaan paketin TCP-yhteys ei lähetä uusia paketteja ennen kuin kadonnut paketti on saatu uudelleenlähetettyä ja onnistuneesti perille. Tämä pakottaa uuden datan odottamaan jo vanhentuneen datan perille pääsyä. Pelikäytössä tarvitaan erilaista luotettavaa tiedonsiirtoa. Sen sijaan että pakettien olisi tultava aina perille ja perille saapuessaan oltava oikeassa järjestyksessä, tarvitaan vain vahvistus jokaisesta saapuneesta paketista. Jos paketista ei ole saapunut vahvistusta esimerkiksi sekunnin kuluessa lähetyksestä, sen voidaan olettaa hävinneen. Tämä mahdollistaa uuden ja nopeasti vanhenevan tiedon välittömän lähetyksen. Hävinneiden pakettejen uudelleenlähetyksestä voidaan päättää tapauskohtaisesti peliohjelmistossa. (Fiedler a. 2008)

Jotta saapuneesta paketista voitaisiin lähettää ilmoitus, täytyy ohjelmiston pystyä tunnistamaan paketit toisistaan. Tätä varten jokaisen paketin otsikkoon lisätään tunnistenumero. Tunnistenumero voi alkaa esimerkiksi nollasta ja numeroa kasvatetaan jokaisen lähetetyn paketin jälkeen. Tällöin ensimmäinen lähetetty paketti saa tunnistenumerokseen 0 ja sadas lähetetty paketti saa tunnistenumerokseen 99. Myös TCP käyttää kyseistä tekniikkaa pakettien tunnistamiseen. TCP kutsuu tätä numeroa järjestysnumeroksi (engl. sequnce number). (Fiedler a. 2008)

Kun lähetetyt paketit on mahdollista tunnistaa toisistaan tunnistenumeron avulla, on vahvistusviestien lähettäminen mahdollista. Yksinkertaisimmillaan tämä tarkoittaa sitä, että vastaanotetun paketin tunnistenumeron lähetetään paketin lähettäjälle. Jos palvelin vastaa jokaiseen asiakkaalta saamaansa datapakettiin omalla datapaketillaan, voidaan vahvistusnumero liittää esimerkiksi datapaketin otsikkoon kuten tunnistenumerokin. Tällainen järjestelmä ei kuitenkaan toimi, jos palvelin vastaanottaa enemmän viestejä kuin lähettää. Tämä järjestelmä ei ole myöskään kovin luotettava. Jos vahvistuksen sisältävä paketti katoaa, olettaa alkuperäisen paketin lähettäjä paketin kadonneeksi vaikka se todellisuudessa saapuikin perille. (Fiedler a. 2008)

Vahvistusviesteistä saadaan luotettavampia, jos jokainen lähetetty paketti sisältää useamman vahvistusviestin. Ongelmana on kuitenkin se, montako vahvistusviestiä jokaisessa paketissa tulisi olla. Jos palvelin lähettää 10 viestiä sekunnissa ja asiakas 30 viestiä sekunnissa, tarvitaan kolme vahvistusviestiä jokaisessa paketissa. Jos paketit kuitenkin ruuhkautuva tarvitaan pahimmassa tapauksessa noin 6-10 vahvistusta jokaisessa paketissa. Tilanne kuitenkin pahenee jos paketteja katoaa. Onkin turvallisempaa lähettää jokaisessa paketissa liikaa vahvistuksia sen sijaan että niitä lähetettäisiin joissain tapauksissa liian vähän. (Fiedler a. 2008)

Usean vahvistusviestin liittäminen pakettiin kuitenkin kasvattaa paketin kokoa. Jos esimerkiksi jokaisessa paketissa lähetetään edellisen paketin vahvistuksen lisäksi 32 ylimääräistä vahvistusta, siis yhteensä 33 vahvistusta, ja jokainen vahvistus on 4 tavua, vievät vahvistukset yhteesä 132 tavua. Kun ottaa huomioon että UDP:n otsikon koko on 8 tavua kasvattaa tällainen lähestymistapa paketin kokoa aivan liikaa (Javvin 2007, 49). Onkin tehokkaampaa tallentaa ylimääräiset vahvistukset bittikenttään. Jokaisen paketin otsikkoon tuleekin siis sekä vahvistusnumero että vahvistusbittikenttä. Vahvistus toimii seuraavasti: Jos vahvistusnumero on 100 ja bittikentän ensimmäinen bitti on asetettu, sisältää paketti vahvistuksen myös paketille 99. Bittikentän toinen bitti vastaa tällöin pakettia 98 ja niin edelleen. (Fiedler a. 2008)

2.2.3 Latenssin seuranta

Latenssilla tarkoitetaan aikaa, joka datapaketilla kuluu verkossa lähettäjältä vastaanottajalle matkaamiseen. Usein latenssista puhuttaessa puhutaan myös RTT:stä (engl. Round Trip Time). RTT tarkoittaa aikaa joka paketilta kuluu matkata verkossa lähettäjältä vastaanottajalle ja takaisin. Usein RTT on tuplasti latenssi, mutta joissain tapauksissa verkon nopeus saattaa olla asymmetrinen. Verkkomoninpeleistä puhuttaessa sana ”lagi” viittaa yleensä RTT:hen. (Armitage & Claypool & Branch 2006, 69)

Latenssin seuranta on varsin helppoa ja onnistuukin yksinkertaisesti mittaamalla lähetettyjen pakettejen RTT:tä. Paketin RTT:n mittaamiseen tarvitaan lista lähetettyjen pakettejen tunnistenumeroista sekä lähetysajoista. Aina kun paketti lähetetään, lisätään sen lähetysaika sekä tunnistenumero listaan. Aina kun paketin saapumisesta saadaan vahvistus, etsitään aiemmin mainitusta listasta vahvistuksen osoittamalla tunnistenumerolla varustettu lähetysaika ja verrataan sitä vahvistuksen saapumisaikaan. Lähetysajan ja vahvistuksen saapumisajan välinen aika on paketin RTT. (Fiedler a. 2008)

Yhteyden keskimääräisen RTT:n seurannassa pitkän ajan kuluessa on otettava huomioon jitter. Jitter tarkoittaa peräkkäisten pakettien latenssien eroa (Armitage & Claypool & Branch 2006, 69). Jitterin vaikutuksen pehmentäminen pidemmän ajanjakson RTT-arvoa varten onnistuu lisäämällä pidemmän ajanjakson RTT-arvoon prosenttiosuuden yksittäisen paketin RTT:n ja pidemmän ajanjakson RTT-arvon erotuksesta. Tämä erotus lisätään pidemmän ajanjakson RTT-arvoon aina kun uuden paketin RTT lasketaan. Lisäämällä noin 10% RTT-arvojen erotuksesta saavutetaan hyvä lopputulos. (Fiedler a. 2008)

2.3 Windows sockets

Windows Sockets (tästä eteenpäin ”Winsock”) on Microsoftin Windows-käyttöjärjestelmille kehittämä ohjelmointirajapinta. Winsock API muistuttaa lähes täydellisesti Berkeley Sockets -ohjelmointirajapintaa. Winsocks on myös yhteentoimiva Berkeley Sockets API:n kanssa. Winsock API:lla toteutettu palvelin voikin ottaa vastaan pyyntöjä Berkeley Sockets API:lla toteutetulta ohjelmistolta jota ajetaan tietokoneelta joka ei käytä Windows-käyttöjärjestelmää. (Hart 2010, 411-412)

Winsock API:n toteutus poikkeaa Microsoftin normaaleista nimeämiskäytännöistä. Tällä on pyritty noudattamaan alan standardeja ja pysymään näin yhdenmukaisena Berkeley Sockets API:iin. Winsock tarjoaa myös standardista poikkeavia ominaisuuksia. Näitä ominaisuuksia tulisi kuitenkin käyttää vain kun se on ehdottoman välttämätöntä. Winsock API ei myöskään ole osa Windows API:a. (Hart 2010, 412)

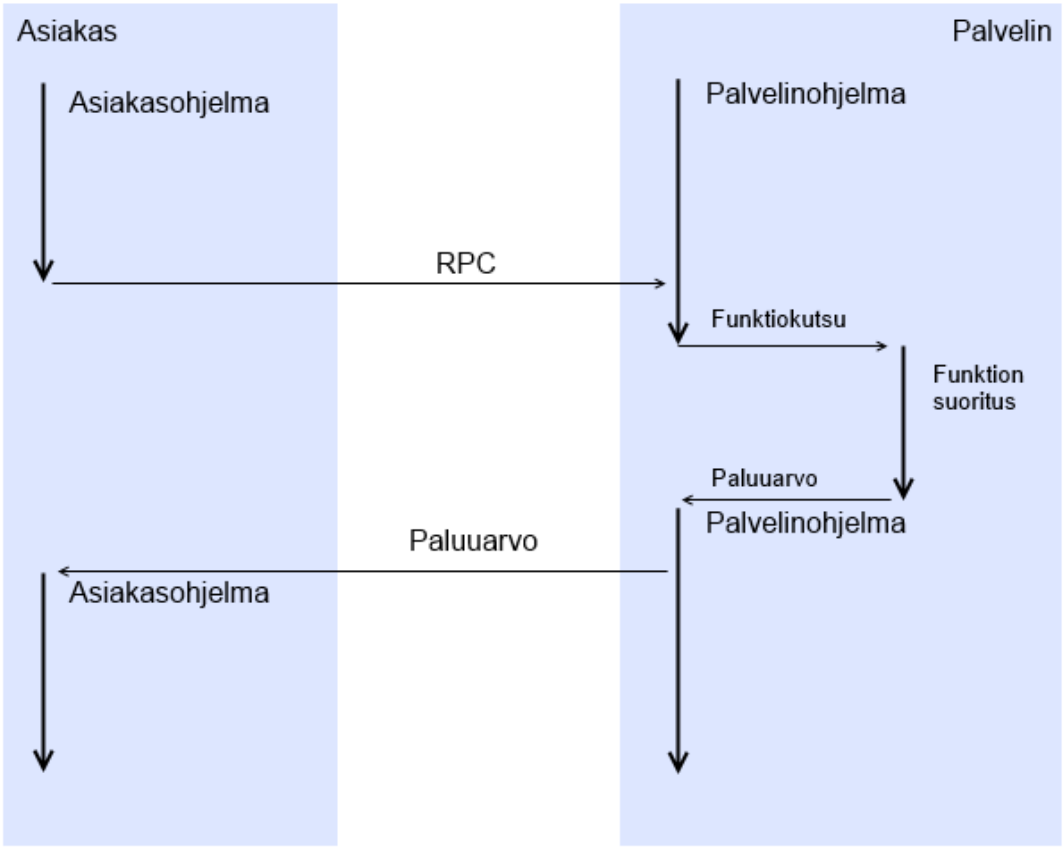
2.4 Prosessien väliset funktiokutsut

Prosessien välinen kommunikaatio tarkoittaa datan siirtämistä prosessien välillä. Datan siirto voidaan suorittaa jaetulla muistilla, mikä mahdollistaa yksinkertaisesti useiden prosessien lukevan ja kirjoittavan erityiseen muistipaikkaan. Prosessien välinen kommunikaatio on mahdollista myös useilla muilla tavoilla, mutta yleensä sitä suoritetaan pistokkeilla. Tämä tarkoittaa, että pakettien lähettäminen ja vastaanottaminen TCP:llä sekä UDP:llä on prosessien välistä kommunikaatiota. (Mitchell & Samuel & Oldham 2001, 95-96, 117)

Tässä luvussa keskitymme prosessien välisiin funktiokutsuihin. Kuten nimestä saattaa päätellä, prosessien väliset funktiokutsut ovat prosessien välistä kommunikaatiota, joka mahdollistaa funktion kutsumisen toisesta prosessista käsin. Tässä opinnäytteessä tutustun kahteen standardiin, jotka mahdollistavat prosessien väliset funktiokutsut. Nämä standardit ovat RPC ja CORBA.

2.4.1 RPC

RPC (Remote Procedure Call) on asiakkaan (engl. client) palvelimelle (engl. server) lähettämä funktiokutsu. Asiakas toimittaa palvelimelle funktion tarvitsemat parametrit ja jää odottamaan palvelimen vastausta. Palvelin suorittaa funktion annetuilla parametreilla ja palauttaa asiakkaalle paluuarvon. Saatuaan paluuarvon palvelimelta asiakas jatkaa toimintaansa normaalisti. RPC:n vaiheet ovat kuvattuna kuviossa 3. Vaikka edellä mainitussa toimintatavassa vain toinen prosesseista on aktiivisena kerrallaan, RPC-protokolla ei rajoita prosessien samanaikaista toimintaa. RPC-protokollan toteutus voi esimerkiksi mahdollistaa asynkroniset RPC-kutsut. Tällöin asiakasohjelma voi suorittaa RPC-kutsusta riippumatonta työtä odottaessaan vastausta RPC-kutsuun. (Lengyel 2010, 341)

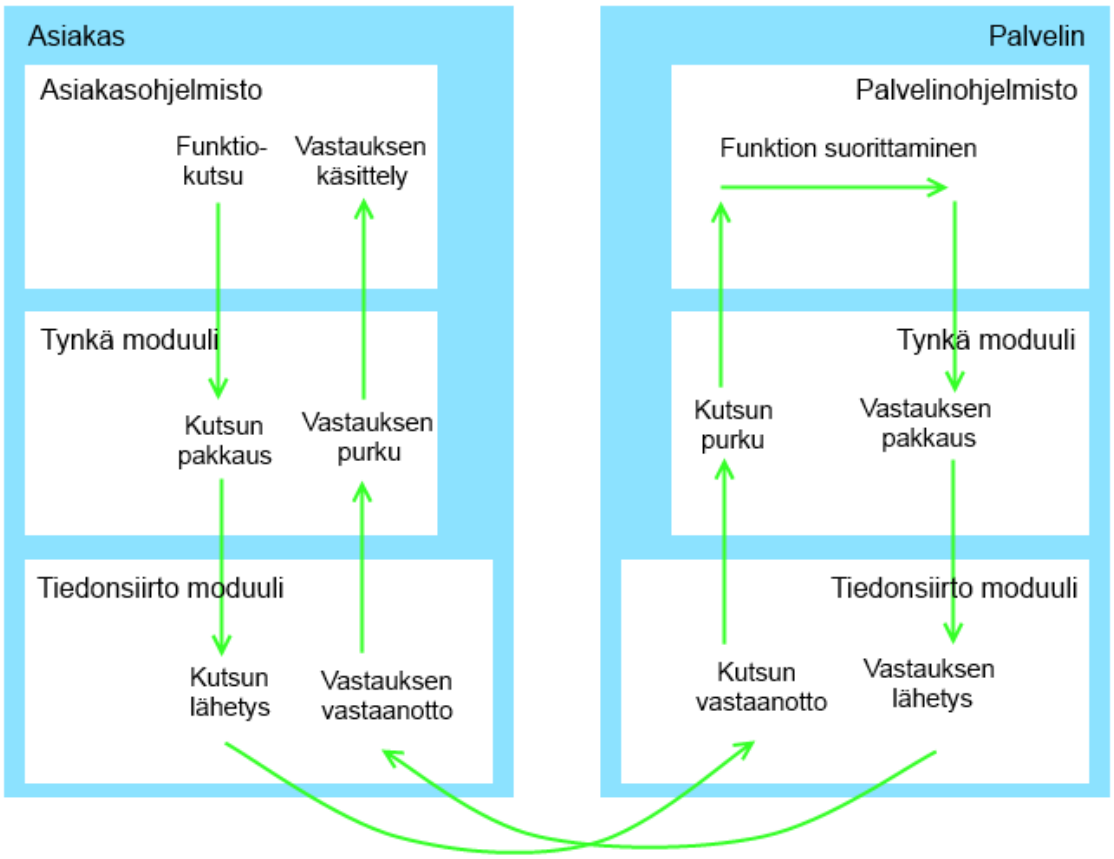


Kuvio 3. RPC:n vaiheet asiakkaalla ja palvelimella (Mukaillen Lengyel 2010, 342).

Jokainen RPC-kutsu sisältää kolme etumerkitöntä kokonaislukuarvoa. Nämä arvot ovat ohjelmiston numero, ohjelmiston versionumero ja funktion numero. Näitä kolmea arvoa käytetään kutsuttavan funktion tunnistamiseen. Ohjelmiston numerot on jaettu kuuteen ryhmään. Näistä ensimmäiseen ryhmään kuuluvia numeroita jakaa IANA ja niitä käyttävät yleisessä käytössä olevat ohjelmistot. Toiseen ryhmään kuuluvia numeroita käyttävät ohjelmistot jotka ovat yksityisessä käytössä. Kolmannen ryhmän numeroita käyttävät ohjelmistot, jotka luovat ohjelmiston numeronsa dynaamisesti. Viimeisten kolmen ryhmän numerot ovat varattuja tulevaisuutta varten. Ohjelmiston versionumeroa käytetään ohjelmiston eri versioiden tunnistamiseen toisistaan. Tällöin ei ole tarpeellista hankkia uutta ohjelmiston numeroa aina kun ohjelmistoon tehdään muutoksia. Funktion numero on kutsuttavan funktion tunnistenumero ja nämä numerot ovat ohjelmisto kohtaisia ja ne tulisi dokumentoida ohjelmiston dokumentaatiossa. (Thurlow 2009, 8, 10)

Kun RPC:tä kutsutaan, erityinen RPC tynkämoduuli valmistelee RPC-kutsun lähettämistä varten. Tynkämoduuli myös käsittelee palvelimelle tulleen RPC-kutsun. RPC-funktiokutsu asiakasohjelmistossa ohjataan tynkämoduulille, joka tunnistaa halutun funktion ja pakkaa

sille annetut parametrit verkkoon lähetettävään pakettiin. Palvelimella tynkäfunktio vastaanottaa RPC-viestin verkosta ja purkaa aliohjelmalle tarkoitetut parametrit vastaanotetusta paketista. Tämän jälkeen tynkämoduuli tunnistaa funktion ja syöttää sille vastaanotetut parametrit. Kun palvelin on suorittanut halutun funktion sen paluuarvon lähetys ja vastaanotto tapahtuu samoin kuin parametrienkin lähetys ja vastaanotto. RPC-kutsun toiminta on kuvattuna kuviossa 4. (Lengyel 2010, 341-342)



Kuvio 4. RPC-kutsun toiminta (Mukaillen Lengyel 2010, 343).

Kuten edellisessä kappaleessa todettiin, tynkämoduuli huolehtii datan pakkaamisesta sekä purkamisesta. Tämä tarkoittaa sitä, että tynkämoduulin on pystyttävä pakkaamaan saamansa parametrit sellaiseen muotoon, että ne voidaan lähettää verkon yli ja sen jälkeen palauttaa ne pakkaamista edeltäneeseen muotoonsa. Tämän takia tynkämoduulin on pystyttävä huomioimaan järjestelmä kohtaisia eroavaisuuksia kokonais- sekä liukulukujen esityksessä. Etenkin järjestelmien mahdollisesti erilaiset tavujärjestelmät on huomioitava. (Lengyel 2010, 344)

Tynkämoduulin on myös pystyttävä käsittelemään eri tavoin syötettyjä parametreja. Parametreja voidaan syöttää kolmella eri tavalla. Parametri voidaan syöttää muokattavana arvona,

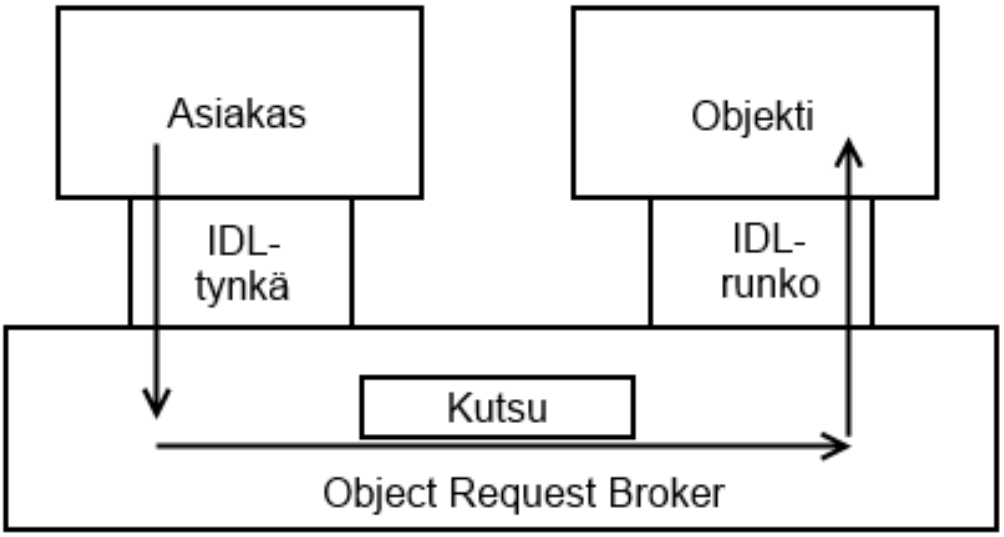
viittauksena tai osoittimena erinäisiin tietorakenteisiin, kuten listoihin tai puumalleihin. Arvona syötetty parametri ei tarvitse erikoistoimia, mutta koska viittauksena sekä osoittimena syötettyt parametrit mahdollistavat alkuperäisarvojen muokkaamisen funktion ulkopuolella, on niiden arvot palautettava muokattuina palvelimelta ja sen jälkeen kopioitava asiakkaan päässä vanhojen arvojen tilalle. (Lengyel 2010, 344)

Koska RPC-järjestelmä mahdollistaa eri järjestelmissä toimivien sovellusten yhdistämisen, mahdollistaa se esimerkiksi pelin ja kehitystyökalujen yhteistoiminnan ajon aikana. Tämä mahdollistaa esimerkiksi virheilmoitusten tulostamisen pelikonsolilla pyörivästä sovelluksesta tietokoneelle. Tietokoneella toimivasta ohjelmasta käsin voidaan myös aktivoida esimerkiksi huijauskoodeja konsolilla pyörivään peliin. Samaan tapaan voidaan mahdollistaa konsolilla pyörivän pelin editointi tietokoneohjelman avustuksella. Tämä helpottaa työskentelyä, koska työkalujen käyttö näppäimistöllä ja hiirellä on huomattavasti helpompaa kuin konsolin ohjaimella. (Lengyel 2010, 347)

2.4.2 CORBA

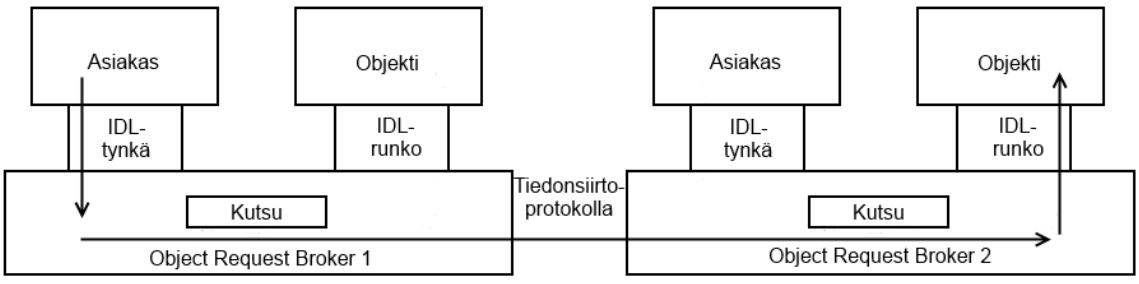
CORBA (Common Object Request Broker Architecture) on Object Management Groupin kehittämä toimittaja riippumaton arkkitehtuuri ja infrastruktuuri, joka mahdollistaa ohjelmistojen yhdessä toimimisen tietoverkkojen välityksellä. CORBA:n standardi protokollaa käyttämällä ohjelmistot voivat toimia yhdessä järjestelmä ja ohjelmointikieli riippumattomasti. (CORBA 2013)

CORBA ohjelmistot koostuvat objekteista. Objektit ovat itsenäisesti toimivia yksiköitä, jotka koostuvat datasta ja toiminnallisuudesta. Yleensä objekteista on myös useita instansseja. Objektit muistuttavatkin oliopohjaisten kielten luokkia. Jokaiselle objekti tyypille määritellään rajapinta OMG IDL-kielellä. Asiakasohjelmistot käyttävät näitä rajapintoja kutsuakseen toimintoja objekteissa. Rajapinta myös hoitaa parametrejen pakkaamisen ja kutsun lähettämisen Object Request Brokerille (ORB). ORB tunnistaa halutun objektin ja lähettää kutsun objektin rungolle. Toiminnon kutsuminen objektista on kuvattuna kuviossa 5. Kun kutsu saavuttaa objektin, IDL-rajapinta huolehtii parametrejen purkamisesta ja toiminnon kutsumisesta itse objektissa. Paluuarvojen lähettäminen takaisin kutsujalle tapahtuu samalla periaatteella IDL-rajapinnan kautta. (CORBA 2013)



Kuvio 5. Toiminnon kutsuminen objektista (Mukaillen CORBA 2013).

Edellisessä kappaleessa kuvattiin toiminnon kutsuminen lokaalista objektista. CORBA mahdollistaa myös ei-lokaalit objektit, jotka näyttävä asiakasohjelmistolle täysin samanlaisilta kuin lokaalitkin objektit. Ei-lokaalin toimintokutsun suorittaminen tapahtuu asiakasohjelmistossa identtisesti lokaaliin kutsuun verrattuna. ORB huolehtii ei-lokaalin kutsun kohdeobjektin löytämisestä ja tarvittavien tietojen lähettämisestä tiedonsiirtoprotokollaa hyväksikäyttäen. Tiedot siirretään halutun objektin lokaalille ORB:lle joka huolehtii kutsun eteenpäin toimittamisesta. Ei-lokaali toimintokutsu on kuvattuna kuviossa 6. (CORBA 2013)



Kuvio 6. Ei-lokaali toimintokutsu (Mukaillen CORBA 2013).

Pelinkehityksessä CORBA:aa voidaan käyttää esimerkiksi chatti-toiminnallisuuden toteuttamiseen. CORBA:lla on luultavasti myös mahdollista toteuttaa geneerinen tiedonvälitysjärjestelmä, joka käyttää XML:ää datan esittämiseen. CORBA mahdollistaa myös hajautetun laskennan peleissä kuten shakki. Kun raskaat siirtojenarviointifunktiot suoritetaan useilla tieto-

koneilla käyttäen CORBA:aa koneiden ohjelmistojen väliseen kommunikaatioon, voidaan saavutta moninkertainen nopeus verrattuna yhdellä tietokoneella suoritettuun arviointiin. (Broekhuizen & Ssekibuule 2004, 9 & Vikram 2005, 45,47)

3 VERKKOKIRJASTO PROJEKTI

Tämän opinnäytetyön käytännön työnä toteutettiin pelinkehitykseen soveltuva verkkokirjasto ja sen testaamiseen käytetty ohjelmisto sekä yksinkertainen peliprojekti. Verkkokirjaston vähimmäistavoitteena oli sen käyttäminen portfolion osana työnhaussa, mutta pyrkimyksenä oli saada kirjastosta niin laadukas että sitä voisi jakaa vapaan lähdekoodin ohjelmistona verkossa.

Henkilökohtaisena tavoitteena tahdoin oppia verkkokirjaston suunnittelusta ja toteuttamisesta mahdollisimman paljon. Tahdoin myös oppia verkko-ohjelmoinnista niin teoriassa kuin käytännössäkin. Tahdoin oppia erityisesti verkko-koodin virheenetsintään ja korjaamiseen liittyviä tekniikoita. Verkkokirjasto toteutettiin kesällä 2013 C++-ohjelmointikielellä ja sen suunnitteluun sekä toteuttamiseen kului hieman yli neljä 40 tunnin työviikkoa. Lähes puolet tästä ajasta kului teorian tutkimiseen sekä projektin suunnitteluun.

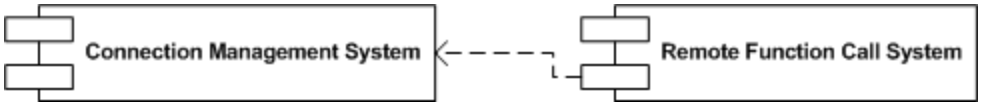
3.1 Vaatimusmäärittely

Päätin verkkokirjaston vaatimukset pääosin itse, mutta keskustelimme verkkokirjaston ominaisuuksista opinnäytetyöni ohjaajan kanssa. Prosessien välisten funktiokutsujen mahdollistaminen verkkokirjaston avulla olikin ohjaajani Mikko Romppaisen ehdotus. Lopulta päätin, että tahdon toteuttaa prosessien väliset funktiokutsut käyttämättä IDL:ää (engl. Interface Definition Language) ja IDL-tiedostoja.

Verkkokirjaston pääasiallinen vaatimus on soveltuvuus pelinkehitykseen. Tämä vaatimus määritteli myös useat muut kirjaston vaatimuksista. Tärkeimpänä näistä on UDP:n käyttö. Tahdoin myös mahdollistaa yhteydellisen tiedonsiirron, sekä kunnollisen järjestelmän yhteyksien hallintaan. Luotettavan tiedonsiirron vaatimuksena pidän sitä, että paketin vastaanottaja tietää missä järjestyksessä paketit on lähetetty ja että lähettäjä saa tiedon jos paketit eivät saavu vastaanottajalle. Pelinkehitykseen soveltuvana ominaisuutena halusin myös toteuttaa kirjastoon mahdollisuuden RTT:n seuraamiselle.

3.2 Arkkitehtuuri ja toiminta

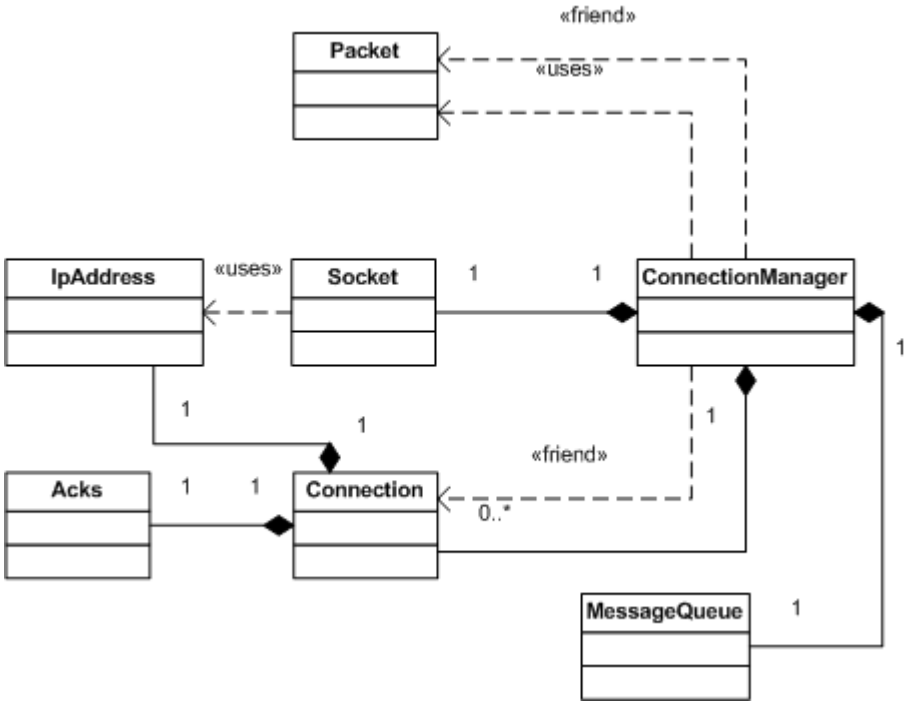
Verkkokirjaston arkkitehtuuri koostuu kahdesta yksinkertaisesta komponentista. Komponenttikaavio on kuvattuna kuviossa 7. Verkkokirjaston pääkomponentti on yhteydenhallintajärjestelmä, joka vastaa virtuaalisten yhteyksien muodostamisesta, ylläpitämisestä sekä lopettamista. Yhteydenhallintajärjestelmä myös lähettää ja vastaanottaa dataa. Verkkokirjaston toinen komponentti vastaa prosessien välisistä funktiokutsuista ja se käyttää datan lähettämiseen yhteydenhallintajärjestelmää. Seuraavissa luvuissa kuvaillaan tarkemmin yhteydenhallintajärjestelmän sekä RFC-komponentin rakennetta ja toimintaa. Luvuissa esitellään myös molempien komponenttien tärkeimmät julkiset funktiot.



Kuvio 7. Verkkokirjaston komponenttikaavio.

3.2.1 Yhteydenhallintajärjestelmä

Yhteydenhallintajärjestelmä koostuu seitsemästä luokasta, jotka ovat kuvattuina kuviossa 8. Yhteydenhallintajärjestelmän sydän on ConnectionManager-luokka, joka vastaa yhteyksien hallinnasta. ConnectionManager-luokka käyttää Packet-luokkaa apuna datapakettien kokoamisessa lähettämistä varten. ConnectionManager-luokalla on yksi Socket-luokan instanssi, joka vastaa tiedon lähettämisestä ja vastaanottamisesta. Socket-luokka käyttää IpAddress-luokkaa apuluokkana IP-osoitteiden hallintaan. ConnectionManger-luokalla on myös lista Connection-luokan instansseja, joista jokainen vastaa yhtä olemassa olevaa virtuaalista yhteyttä. ConnectionManager-luokalla on myös yksi insanssi MessageQueue-apuluokasta, jonka tehtävä on mahdollistaa säieturvallinen kirjoitus ja luku verkosta vastaanotetuille viesteille. Yhteydenhallintajärjestelmä käyttää myös public domainin alaista fastdelegates-järjestelmää (Clugston 2005). Fastdelegates-järjestelmällä on toteutettu yhteydenhallintajärjestelmän käyttämät callback- sekä delegaatti-funktiot.



Kuvio 8. Yhteydenhallintajärjestelmän luokkakaavio.

ConnectionManager-luokka

Vaikka verkkokirjaston Socket-luokkaa voi käyttää itsenäisesti datan lähettämiseen ja vastaanottamiseen, on verkkokirjastoa tarkoitus käyttää ConnectionManager-luokan välityksellä. ConnectionManager-luokka toimiikin rajapintana peliohjelmiston ja verkkokirjaston verkko-ominaisuuksien välillä.

ConnectionManager-luokan kuusi tärkeintä funktiota ovat Initialize, Shutdown, ConnectTo, DisconnectClient, Send ja GetPacket. Nimensä mukaisesti Initialize-funktio alustaa tarvittavat muuttujat, sitoo halutun pistokkeen porttiin, varaa muistia sekä käynnistää datan vastaanottamisesta sekä ajastinten päivittämisestä vastaavat säikeet. Vastaavasti Shutdown-funktio vapauttaa varatun muistin, vapauttaa sidotun pistokkeen sekä sulkee säikeet. ConnectTo-funktio yrittää luoda yhteyden haluttuun IP-osoitteeseen sekä porttiin ja vastaavasti DisconnectClient-funktio sulkee yhteyden haluttuun kohteeseen. Send- ja GetPacket-funktiot vastaavat datan lähettämisestä ja vastaanottamisesta. Send-funktiolla dataa voidaan lähettää joko Packet-luokan avulla tai funktiolle voi syöttää parametrina osoittimen dataan, jonka ConnectionManager-luokka pakkaa automaattisesti Packet-luokan instanssiin lähettämistä varten.

Suurin osa ConnectionManager-luokan muista käyttäjälle paljastetuista funktioista asettaa erilaisia callback-funktioita. Callback-funktio on kutsuja luokan, tässä tapauksessa ConnectionManager-luokka, ulkopuolinen funktio, jota kutsutaan tietyn ehdon täyttyessä. ConnectionManager-luokkaan voidaan asettaa callback-funktiot seuraaville tilanteille: uusi yhteys on luotu onnistuneesti, uuden yhteyden muodostaminen on epäonnistunut, asiakas on katkaissut yhteyden, yhteys on katkennut tuntemattomasta syystä ja lähetetty datapaketti ei ole saapunut perille. ConnectionManager-luokalle voi myös asettaa delegaatti-funktion, jolle ohjataan prosessien välisen funktiokutsun tiedot sisältävät paketit.

ConnectionManager-luokalta pystyy myös selvittämään IP-osoitetta vastaavan yhteystunnistenumeron sekä yhteystunnistenumeroa vastaavan yhteyden tiedot. ConnectionManager-luokalta pystyy pyytämään myös yksittäisen yhteyden RTT:n. ConnectionManager-luokka mahdollistaa myös IP-osoitteiden lisäämisen mustalle listalle. Mustalla listalla olevilta IP-osoitteilta tulevat yhteyspyynnöt jätetään huomiotta.

ConnectionManager-luokka käyttää Socket-luokkaa datapakettejen lähettämiseen ja vastaanottamiseen. MessageQueue-luokan instanssi vastaa vastaanotettujen datapakettejen säilöämisestä kunnes GetPacket-funktiota kutsutaan. MessageQueue on toteutettu säieturvallisesti. Kaikki ConnectionManager-luokan välityksellä lähetettävä data pakataan Packet-luokan instansseihin ja ConnectionManager-luokka asettaa automaattisesti omaa tunniste- sekä seurantadataansa jokaisen datapaketin otsikkoon. ConnectionManager käyttää Connection-luokan instansseja pitääkseen kirjaa kaikista muodostetuista yhteyksistä.

Socket-luokka

Socket-luokka tarjoaa verkkokirjaston tärkeimmät toiminnallisuudet, eli verkkopistokkeen sitomisen sekä datan lähettämisen ja vastaanottamisen. Socket-luokan Bind- ja Unbind-funktioilla sidotaan ja vapautetaan pistoke. Bind-funktiolle määritetään halutun portin numero. Socket-luokalta voi myös tiedustella GetPort-funktiolla nykyisen portin numeron.

SetBlocking-funktiolla voidaan asettaa pistoke joko estävään tai estämättömään tilaan. Estävässä tilassa Receive-funktio estää ohjelman jatkumisen, kunnes se vastaanottaa dataa. Estämättömässä tilassa Receive-funktio vastaavasti palaa funktiosta vaikka dataa ei olisikaan saapunut. Socket-luokka on oletuksena estävässä tilassa. Socket-luokan tilaa voi tiedustella IsBlocking-funktiolla.

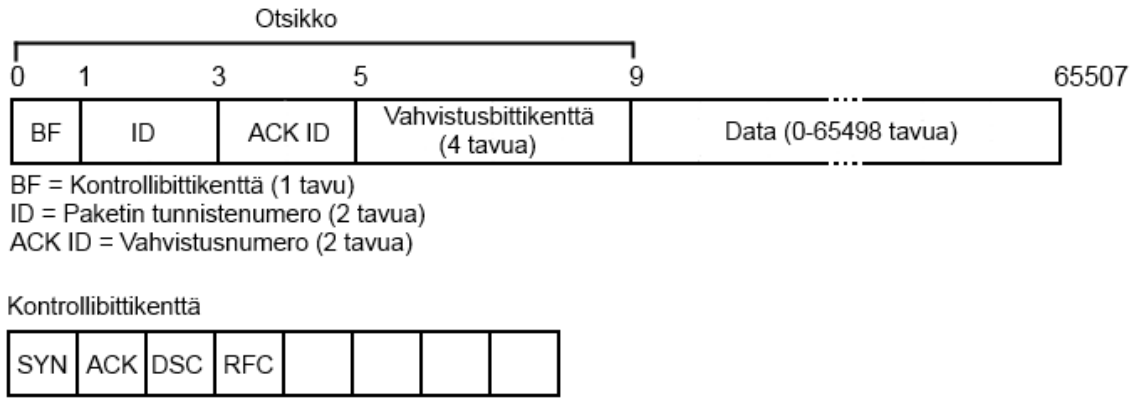
Socket-luokan yleisimmin käytetyt funktiot ovat Send ja Receive. Send-funktiolla lähetetään ja Receive-funktiolla vastaanotetaan dataa. Molemmilla funtiolla on mahdollista käsitellä dataa joko osoittimena datapuskuriin tai viittauksena Packet-luokan instanssiin. Send-funktiolle annetaan parametreina vastaanottajan IP-osoite sekä portti, mutta Receive-funktio käsittelee kaiken pistokkeen sen hetkiseen porttiin saapuvan datan. Receive-funktio tallentaa datan lähettäjän IP-osoitteen sekä portin muuttujaparametreihin.

Packet-luokka

Packet-luokkaa käytetään datan pakkaamiseen lähettämistä varten. Packet-luokka myös sisältää ConnectionManager-luokan käyttämän otsikon ja sen muokkaamiseen tarvittavat funktiot. Packet-luokan tärkein ominaisuus on mahdollisuus lukea ja kirjoittaa Packet-luokan instanssiin kaikkia perustietotyyppejä sekä niiden lyhyitä ja pitkiä versioita. Packet-luokka tukee myös string-luokan kirjoittamista ja lukemista. Packet-luokka koostuu yhdeksän tavun kokoisesta otsikosta sekä maksimissaan 65 498 tavun kokoisesta data osiosta. Data osio voi tarvittaessa olla tyhjä.

Packet-luokan dataan lisäämä otsikko on yhdeksän tavun kokoinen. Otsikon ensimmäinen tavu sisältää kontrollibittikentän. Bittikentän neljää ensimmäistä bittiä käytetään viestin tyypin tunnistamiseen. Ensimmäinen bitti on SYN bitti, joka on asetettu kun viesti koskee uuden yhteyden luomista. Toinen bitti, eli ACK bitti, on asetettuna kun viesti on positiivinen vastaus yhteyspyyntöön. Kolmas bitti on DSC, eli disconnect, bitti, joka on asetettuna yhteyden lopettamista koskevissa viesteissä. Neljäs bitti on RFC bitti joka ilmoittaa paketin olevan prosessien välinen funktiokutsu. Bittikentän loput neljä bittiä on varattu verkkokirjastoa käyttävän ohjelmiston tarpeisiin. Bittikenttää käytetään SetControlBit-, ClearControlBit- sekä CheckControlBit-funktioiden avulla.

Bittikentän jälkeen otsikossa on kahden tavun kokoinen paketin tunnistenumero. Tästä seuraavat kaksi tavua ovat vahvistusnumero ja sitä seuraa neljän tavun kokoinen vahvistusbittikenttä. Vahvistusnumeron ja vahvistusbittikentän toiminta on kuvattuna luvussa 2.2.2. Tunnistenumero asetetaan SetPacketId-funktiolla ja se voidaan noutaa GetPacketId-funktiolla. Vahvistustietojen käsittely onnistuu SetAckId-, GetAckId-, SetAckBitfield- sekä GetAckBitfield-funktioilla. Paketin rakenne on kuvattuna kuviossa 9.



Kuvio 9. Paketin rakenne.

Packet-luokka tarjoaa funktiot myös kaiken paketin sisältämän datan noutamiselle sekä asettamiselle. GetData-funktio palauttaa osoittimen paketin sisältämään dataan. Tämä data ei sisällä otsikkoa. GetDataWithHeader-funktio palauttaa osoittimen paketin sisältöön sekä otsikkoon. Otsikko sijaitsee puskurissa ennen itse dataa. Vastaavasti GetDataSize-funktio palauttaa pelkän datan koon tavuina ja GetDataSizeWithHeader-funktio palauttaa sekä datan että otsikon koon tavuina. SetData- ja SetDataAndHeader-funktiot mahdollistavat nimensä mukaisesti paketin datan sekä paketin datan ja otsikon asettamisen manuaalisesti.

3.2.2 RFC-komponentti

RFC-komponentti mahdollistaa prosessien välisten funktiokutsujen suorittamisen ja se on toteutettu yhdellä luokalla sekä kokoelmalla makroja. RFC-komponentti käyttää yhteydenhallintajärjestelmää funktiokutsujen välittämiseen, mutta yhteydenhallintajärjestelmän korvaaminen jollain muulla tiedonvälitystavalla on varsin helppoa. Tiedonvälitysjärjestelmä tarvitsee vain boolean-arvon palauttavan viestinlähetysfunktion, joka ottaa vastaan parametreina lähetettävän paketin sekä vastaanottajan tunnistenumeron. Vastaavasti viestinvälitysjärjestelmään täytyy lisätä funktio-osoitin RFC-komponentin HandleMessages-funktiolle, joka vastaa viestien käsittelystä.

RFC-komponentti mahdollistaa maksimissaan kahdeksan parametria sekä paluuarvon sisältävien funktioiden käytön prosessien välisinä funktioina. Kahdeksan parametrin rajoitus tulee Don Clugstonin kehittämästä fastdelegates-järjestelmästä (Clugston 2005). RFC-komponentti käyttää Clugstonin järjestelmää funktio-osoittimien käsittelyyn. RFC-

komponentti mahdollistaa kaikkien perus tietotyyppien sekä itse tehtyjen luokkien käyttämisen parametreina sekä paluuarvona. Osoittimien ja viittausten käyttö ei ole tuettuna. Itse tehtyjen luokkien on ylikuormitettava syöttö- ja tulostusoperaattorit (<< ja >>) Packet-luokalle, jotta niiden käyttö RFC-komponentin kanssa olisi mahdollista. RFC-komponentti käyttää fastdelegates-järjestelmän lisäksi myös toista public domainin alaista järjestelmää. Austin Applebyn MurmurHash3-järjestelmää käytetään funktioiden tunnistenumerojen luomiseen (Appleby 2011).

RFC-komponentti on toteutettu RFCSystem nimisenä singleton-luokkana, eli RFC-komponentista voi olla olemassa vain yksi instanssi. Singleton käyttäytyminen on toteutettu staattisella GetInstance-funktiolla, joka palauttaa osoittimen RFC-komponentin singleton instanssiin, joka on RFCSystem-luokan yksityinen tietojäsen. Jos singleton tietojäsentä ei ole vielä alustettu, GetInstance-funktio alustaa sen ennen osoittimen palauttamista. Singleton instanssin voi tuhota kutsumalla staattista Reset-funktiota. Reset-funktio tuhoaa RFCSystem-luokan singleton instanssin, joten kun GetInstance-funktiota kutsutaan seuraavan kerran alustetaan uusi singleton instanssi.

RFCSystem-luokalla on kolme rekisteröinti funktiota. RegisterSendMessageDelegate-funktio rekisteröi viestien lähettämiseen käytettävän delegaatti funktion. Kuten aikaisemmin mainittiin, viestinlähetysfunktion täytyy ottaa parametreina lähetettävä data Packet-luokan instanssina sekä yhden tavun kokoinen vastaanottajan tunnistenumero. Viestinlähetys funktion täytyy myös palauttaa tieto lähettämisen onnistumisesta boolean-arvona. RegisterCallFailedCallback-funktio rekisteröi callback-funktion, jota kutsutaan jos prosessien välinen funktiokutsu epäonnistuu. Callback-funktion on vastaanotettava parametreinä kutsutun funktion tunnistenumero sekä kutsun tunnistenumero. RegisterFunction-funktio rekisteröi prosessien välisen funktion. Tätä funktiota kutsutaan normaalisti vain makron välityksellä.

RFC-komponentin kaksi useiten kutsuttua funktiota ovat CallFunction sekä HandleMessages. Kuten aiemmin todettiin HandleMessages-funktio vastaa viestien käsittelystä. CallFunction-funktio suorittaa prosessien välisen funktiokutsun halutulle funktiolle annetuilla parametreillä. Tätä funktiota kutsutaan normaalisti vain makrolla luodusta tynkäfunktiosta.

RFC-järjestelmä käyttää makroja tynkäfunktioiden luomiseen sekä prosessien välisten funktioiden rekisteröintiin RFC-järjestelmään. Makrot noudattavat yksinkertaista nimeämiskäytäntöä käytön helpottamiseksi. CREATE_STUBSX nimisellä makrolla luodaan tynkäfunkti-

ot prosessienvälisellefunktiolle. Makron nimessä kirjain X korvataan numerolla 0-8, jolloin numero tarkoittaa funtion parametrien määrää. CRETE_STUBSX makroista on myös _HEADER loppuiset versiot. Näitä versoita on tarkoitus käyttää otsikkotiedostossa. CREATE_STUBSX_RETURN nimisillä makroilla luodaan tynkä funktiot paluuarvon sisältäville funktioille. Myös näissä makroissa X-kirjain korvataan halutulla parametrien määrällä ja otsikkotiedostoja varten on olemassa _HEADER loppuiset versiot. Samaa nimeämiskäytäntöä noudattaen prosessien välisten funktioiden rekisteröintiä varten on olemassa REGISTER_DELEGATEX sekä REGISTER_DELEGATEX_RETURN makroja. Makrot ottavata parametreina pääsääntöisesti makroon liittyvän funtion nimen sekä parametrien ja mahdollisen paluuarvon tietotyypit. Tynkienluontimakrot ottavat parametreinä myös kutsun aikakatkaisun pituuden sekunteina sekä parametrien ja mahdollisen paluuarvon koon tavuina.

3.3 Toteutus

Opinnäytetyöprojekti aloitettiin kesäkuun alussa 2013 ja itse verkkokirjaston ohjelmointi aloitettiin elokuun puolessavälissä. Verkkokirjasto ohjelmoitiin käyttäen Microsoft Visual Studio 2010 ohjelmointiympäristöä sekä C++-ohjelmointikieltä. Versionhallintaan käytettiin Bitbucket palvelua sekä SmartGit/Hg git palvelinohjelmistoa. Projektin aikataulutukseen sekä tehtävälistan ylläpitoon käytettiin AgileZen palvelua.

Projekti aloitettiin tutustumalla verkko-ohjelmointiin, sekä pelinkehityksessä käytettyihin ja vaadittuihin matalan tason toiminnallisuuksiin. Verkkokirjaston kehittämiseen tarvittavaa tietoa kerättiin heinäkuun loppuun asti. Aikaa tiedon keräämiseen käytin noin kaksi tuntia jokaisena arkipäivänä.

Itse verkkokirjasto projekti alkoi elokuussa viikon suunnittelulla. Suunnittelu viikon aikana tein verkkokirjastolle vaatimustenmäärittelyn, valmistelin git palvelimen projektia varten sekä suunnittelin verkkokirjaston arkkitehtuurin sekä tein alustavat luokkakaaviot. Tein suunnittelun aikana myös useita testejä esimerkiksi aiemmin mainitun fastdelegates-järjestelmän toiminnasta sekä tynkäfunktioiden luomisesta makrojen avulla. Näillä testeillä sain lyötyä lukkoon kriittisimpien ja itseäni eniten askarruttaneiden ominaisuuksien toteutus tavan.

Elokuun puolessavälissä aloitin verkkokirjaston ohjelmoinnin. Samalla siirryin 6-8 tunnin työpäiviin aikaisemmin käyttämieni lyhyempien työpäivien sijaan. Aloitin verkkokirjaston

toteutuksen IpAddress-luokasta, jonka sainkin toimivaksi jo ensimmäisenä työpäivänä. Tässä vaiheessa aloitin myös ohjelmoimaan yksinkertaista verkkokirjaston ominaisuuksien testaamiseen käytettävää komentorivisovellusta. Laajensin tätä testisovellusta sitä mukaa kun sain toteutettua uusia ominaisuuksia verkkokirjastoon. Testasinkin jokaisen uuden toiminnallisuuden välittömästi sen lisäämisen jälkeen. Ensimmäisenä päivänä aloitin myös Socket-luokan toteuttamisen.

Projektin kolmantena päivänä olin toteuttanut IpAddress- ja Socket-luokkien lisäksi myös Packet-luokan. Pystyinkin tässä vaiheessa lähettämään Packet-luokan avulla järjestettyä dataa sekä raakadataa kahden ohjelmiston välillä. Verkkokirjasto olikin huomattavasti edellä aikataulua eikä toteutuksessa ollut tullut vastaan minkäänlaisia ongelmia. Testattuani omalla koneellani edellä mainitut luokat siirryin toteuttamaan ConnectionManager-luokkaa.

Seuraavat kaksi päivää menivät lähinnä hyvin vaikeasti löydettäviä virheitä etsiessä ja korjatessa. ConnectionManager-luokan Receive-funktio esimerkiksi tuntui estävän vaikka pistoke olikin asetettu estämättömään tilaan. Ongelman syyksi paljastui pitkällisen tutkimisen jälkeen se, että pistoke asetettiin estämättömään tilaan vasta sen jälkeen, kun Receive-funktiota oli jo kutsuttu. Tällöin pistoke oli estävässä tilassa ja kuunteli vastaanotettavaa dataa kun pistokkeen tilaa yritettiin muuttaa. Vaikka pistokkeen estämistilaa muuttava funktio palauttaakin onnistumista kuvaavan arvon pistokkeen jo kuunnellessa tulevaa dataa, ei pistoke muutu todellisuudessa estämättömäksi ennen kuin se on palannut datankuuntelufunktiosta seuraavan kerra. Ongelma korjaantui yksinkertaisesti asettamalla pistoke estämättömäksi ennen datan kuuntelusta vastaavan säikeen käynnistystä.

Seuraavat kolme päivää sujuivat varsin kivuttomasti ja sainkin yhteydenhallintajärjestelmän valmiiksi sekä kommentoin ja siivosin koodin. Käytinkin yhden päivän koodin siistimiseen ja yleiseen paranteluun. Kirjoitin myös kattavampia testejä yhteydenhallintajärjestelmälle. Olin projektin alusta lähtien kiinnittänyt erityistä huomiota koodin laatuun ja se helpottikin koodin lopullista siistimistä. Koodin helppo luettavuus oli myös helpottanut virheiden löytämistä projektin aiemmissa vaiheissa. Koodin siistimisen jälkeen oli myös helpompaa kirjoittaa testitapauksia ohjelmistolle.

Toteutin RFC-komponentin seuraavan neljän päivän aikana. Suurimmaksi ongelmaksi muodostuivat RFC-järjestelmän käyttämät lukuisat makrot. Makrojen virheiden etsintä ja korjaus onkin erittäin vaikeaa, koska Visual Studion debuggeri ei pysty näyttämään makrojen sisältöä

koodia rivi kerrallaan läpi käytäessä. Myöskään Visual Studion Intelli-sense järjestelmä ei osaa luotettavasti ilmoittaa makrojen sisällä olevista kirjoitus- tai syntaksivirheistä. RFC-järjestelmän valmistuttua laajensi testiohjelmistoani sekä suoritin testin internetin välityksellä ystäväni avustuksella. Verkkokirjaston suunnitteluun ja toteutukseen, tiedonkeruu mukaan luettuna, kului loppujen lopuksi noin neljä 40 tunnin työviikkoa.

3.4 Testaus

Verkkokirjastoa testattiin yksinkertaisella testiohjelmistolla aina, kun verkkokirjastoon lisättiin uusi ominaisuus tai vanhaa koodia muutettiin. Testaus suoritettiin konsolisovelluksella, josta samalla tietokoneella pyöritettiin sekä palvelin- että asiakasohjelmistoja. Testitapaukset kirjoitettiin niin, että yksittäisten ominaisuuksien testit olivat omissa testitapauksissaan, vaikka seuraava testitapaus olisikin voinut hyväksikäyttää esimerkiksi edellisessä testitapauksessa luotua ConnectionManager-luokan instanssia. Tällöin yhden testitapauksen epäonnistuminen ei vaikuttanut seuraavien testitapauksien tuloksiin. Myöskin aina testi ohjelmaa pyöritettäessä ajettiin myös kaikki vanhat testit. Näin ollen oli mahdollista huomata, jos uusi ominaisuus oli rikkonut vanhemman ominaisuuden.

Verkkokirjaston tärkein vaatimus oli, että se käyttää UDP:tä tiedonsiirtoon. Ominaisuuden testaaminen oli varsin yksinkertaista. Testiohjelma vain lähetti dataa UDP-pistokkeen kautta toiseen sovellukseen. Kuten kaikissa muissakin testeissä, kehityksen aikana testit ajettiin yhdellä tietokoneella käyttäen useaa yhtäaikaa pyörivää sovellusta ja vasta koko verkkokirjaston valmistuttua testi ajettiin kaikkien muiden testien kanssa verkon yli. UDP-testi todettiin onnistuneeksi kun lähetetty data saapui onnistuneesti perille. Koska tiesin mitä lähetetty data sisälsi, oli minun helppoa tarkistaa datan aitous vastaanottavassa sovelluksessa.

Yhteydellistä tiedonsiirtoa testattiin yhdellä tietokoneella pyörittäen yhtä palvelinsovellusta, johon useat asiakassovellukset ottivat yhteyden. Yhteyksien luonnin jälkeen palvelin lähetti jokaiselle asiakassovellukselle viestin, johon asiakkaat vastasivat. Tarkistin palvelimella, että viestit olivat tulleet oikeilta asiakkailta vertaamalla viestien sisältöä. Tämä oli helppoa, koska palvelin lähetti jokaiselle asiakkaalle yksilöllisen viestin ja asiakkaat vastasivat viestiin lähettämällä saman viestin takaisin.

Kuten vaatimusmäärittelyssä mainitsin, pidän tässä tapauksessa luotettavan tiedonsiirron vaatimuksena sitä, että vastaanottaja tietää missä järjestyksessä paketit on lähetetty ja että lähettäjä tietää jos paketti ei ole saapunut vastaanottajalle sovitun ajan kuluessa. Lähetettyjen pakettien järjestys testattiin lähettämällä asiakkaalle peräkkäin useita paketteja, joiden data oli ainoastaan paketin järjestysnumero. Tämä järjestysnumero oli sama jonka ConnectionManager-luokka lisää automaattisesti jokaisen paketin otsikkoon lähetyksen yhteydessä. Pakettien vastaanottaja pystyi varmistumaan pakettien otsikoissa olevan järjestysnumeron oikeellisuudesta vertaamalla sitä paketin datassa olevaan järjestysnumeroon.

Hävinneistä paketeista ilmoittamista testattiin lähettämällä asiakkaalle useita paketteja ja estämällä asiakasta vastaamasta paketteihin. Tämän jälkeen palvelinohjelmisto odotti pakettien aikakatkaisuksi asettamansa ajan ja jos pakettien häviämisestä ilmoittavaa callback-funktiota kutsuttiin testi todettiin onnistuneeksi.

RTT:tä testattiin lähettämällä useita paketteja palvelimelta asiakkaalle ja vastaamalla asiakkaalla saatuihin viesteihin joko välittömästi tai sovitun viiveen jälkeen. Tämän jälkeen palvelimella verrattiin yhteydenhallintajärjestelmän ilmoittamaa RTT:tä oletettuun RTT:hen. Jos arvot olivat lähellä toisiaan, testi todettiin onnistuneeksi. En käyttänyt testissä kiinteitä rajaarvoja, vaan tulostin oletetun sekä yhteydenhallintajärjestelmän ilmoittaman ajan ja vertasin niitä manuaalisesti keskenään. Tämä hidasti testin tekemistä, mutta antoi minusta tarpeeksi luotettavia tuloksia. Pystyin tällöin myös ottamaan huomioon ohjelmistoista itsestään johtumattomat viiveet, jotka pystyin huomaamaan katsomalla vierekkäin sekä palvelin- että asiakasohjelmistojen toimintaa ja niiden tulostamia ilmoituksia.

Prosessien välisiä funktiokutsuja testasin kirjoittamalla testifunktiot jokaiselle RFCkomponentin makrolle. Käytin mahdollisuuksien ja tarpeen mukaan toisistaan eroavia paluu sekä parametri tyyppejä. Testit olivat kutsujen osalta onnistuneita, jos palvelin vastaanotti samat parametrien arvot kuin asiakas oli lähettänyt. Arvojen vertaaminen oli helpoa koska tiesin ne etukäteen. Kutsujen paluuarvot olivat onnistuneita, jos kutsut palauttivat oletetut arvot. Tämä oli helppo tarkistaa, koska funktioiden toiminta oli erittäin yksinkertaista, lähinnä yhteenlaskuja.

Kaikki edellä mainitut testit, sekä suuri joukko yksittäisten pienempien ominaisuuksien testejä, toimivat moitteettomasti valmiilla verkokirjastolla. Näiden testejen lisäksi kirjoitin esimerkiksi kirjoitus- ja luku testit kaikille Packet-luokan tukemille tietotyypeille sekä yhdelle itse

kirjoittamalleni luokalle. Testasin kaikkien tietotyyppien sekä oman luokkani toiminnan myös prosessien välisten funktiokutsujen testaamisen yhteydessä.

Edellä kuvatun testiohjelmiston lisäksi kehitin testaamista varten yksinkertaisen pelin. Peli on verkon välityksellä toimiva ammuskelupeli, jossa useat pelaajat yrittävät tappaa toisensa. Pelissä ei voi sinäänsä voittaa eikä siinä ole taivoitteita, koska pelin pelilliset ominaisuudet olivat toissijaisia testaukseen verrattuna.

Peli koostuu palvelinsovelluksesta sekä erillisestä asiakassovelluksesta. Pelaaja käyttää asiakassovellusta, joka ottaa yhteyden palvelinsovellukseen. Palvelinsovellus vastaa pelimekaniikan laskemisesta sekä tiedon välittämisestä kaikille pelaajille. Pelissä on kolme tilaa, valikko, aula sekä peli. Valikkotilassa asiakassovellus antaa pelaajan valita mitä porttia verkkokirjasto käyttää. Palvelimen valikkotila mahdollistaa halutun portin sekä yhtäaikaisten pelaajien enimmäismäärän valinnan. Aulatilassa asiakasohjelmisto kysyy pelaajalta palvelinohjelmiston IP-osoitetta sekä porttia. Palvelimeen yhdistämisen jälkeen asiakassovellus odottaa pelin aloituskäskyä palvelinsovellukselta. Aulatilassa palvelinsovellus odottaa, kunnes kaksi pelaajaa on liittynyt peliin. Tämän jälkeen palvelinsovellus ilmoittaa asiakassovelluksille pelin aloittamisesta. Pelitilassa on toteutettuna edellä manitut pelilliset ominaisuudet. Kuviossa 10 on näkyvissä ruutukaappaus pelitilasta asiakassovelluksessa.

Kuvio 10. Ruutukaappaus pelin pelitilasta asiakassovelluksessa.

Pelillä ei testattu verkkokirjaston ominaisuuksien toimintaa, koska ominaisuuksien testaamiseen käytettiin aikaisemmin kuvattua testiohjelmistoa. Tämän sijaan pelillä testattiin verkkokirjaston soveltuvuutta pelinkehitykseen, eli tarkoituksena oli havaita puuttuuko verkkokirjastosta ominaisuuksia joita peliprojektin kehityksen aikana tarvitaan.

Testauksen seurauksena verkkokirjastoon lisättiin kolme uutta funktioita ja Send-funktioon lisättiin toiminnallisuus. Ensimmäisenä ConnectionManager-luokkaan lisättiin IsRunning-funktio. Funktiolla voi tarkistaa onko yhteydenhallintajärjestelmä käynnissä. Seuraavaksi ConnectionManager-luokkaan lisättin GetMaxConnections- ja GetConnectionCount-funktiot. Ensimmäisellä funktiolla yhteydenhallintajärjestelmältä voi tiedustella yhtäaikaisten yhteyksien enimmäismäärän ja toisella funktiolla voi tiedustella auki olevien yhteyksien määrää. Send-funktioon lisättiin mahdollisuus tallentaa lähetettävän paketin tunnistenumero muuttujaparametriin.

Testaamista varten kehitetty peli oli erittäin hyödyllinen verkkokirjaston käytettävyyden testaamiseen. Testejen aikana huomattiinkin muutamia käytettävyysongelmia, jotka korjattiin lisäämällä edellä kuvaillut funktiot sekä toiminnallisuudet. Tällaisten käytettävyysongelmien toteaminen rajoittuneessa ja vain ominaisuuksien testaamiseen tarkoitetussa sovelluksessa onkin hyvin hankalaa. Tästä johtuen ohjelmistoja tulisikin aina testata niiden oikeassa käyttötarkoituksessa pelkkien rajattujen keinotekoisten testien lisäksi.

3.5 Jatkokehitysajatukset

Verkkokirjasto täyttää nykyisellään mielestäni alussa sille asettamani tavoitteet. Mielestäni makrojen käyttö on huonoa ohjelmointia ja sen takia olisin halunnut toteuttaa RFC-komponentin toiminan jollain elegantimmalla tavalla. En kuitenkaan keksinyt käyttäjän kannalta helpompaa tapaa toteuttaa kyseistä järjestelmää. Pidänkin tärkeämpänä sitä, että verkkokirjaston ominaisuudet ovat helppoja käyttää kuin sitä että koodi on optimaalisen siistiä.

Tulevaisuutta ajatellen haluaisin toteuttaa verkkokirjastoon kaksi muutosta. Ensinnäkin haluaisin lisätä Packet-luokkaan serialisoinnin, joka mahdollistaisi datan lähettämisen eri tavujärjestystä käyttävien tietokoneiden välillä. Jätin ominaisuuden toteuttamatta, koska kaikki saatavillani olevat tietokoneet käyttivät samaa tavujärjestystä. Näin ollen ominaisuuden luotettava testaaminen oli mielestäni mahdotonta. Hyvänä puolena serialisoinnin puuttuminen tekee

verkkokirjastosta marginaalisesti nopeamman ja ominaisuuden voi toteuttaa tarvittaessa kirjaston ulkopuolisestikkin.

Toisena muutoksena tahtoisin tehdä verkkokirjastosta useilla alustoilla toimivan version. Keskityin tekemään verkkokirjaston vain Windows-käyttöjärjestelmälle, koska se säästi aikaa ja käytössäni oli vain Windows-koneita. Tästä johtuen monialustatoiminnalisuuden testaaminen oli minulle tarpeettoman vaikeaa. Riittävän testilaitteiston saatavilla ollessa ominaisuuden toteuttamisen pitäisi olla varsin yksinkertaista. Verkkokirjaston kääntäminen esimerkiksi Unix-ympäristöön olisi yksinkertaista, koska verkkokirjastossa käytetty Winsock-ohjelmointirajapinta on hyvin lähellä muiden alustojen käyttämää standardia ja käytin varsin vähän vain Windowsilla toimivia funktioita.

4 POHDINTA

Verkkokirjastoprojekti onnistui mielestäni yli odotusten. Ensinnäkin sain toteutettua kaikki haluamani ominaisuudet, sekä toteutin projektin huomattavasti suunniteltua nopeammin. Opin myös projektista todella paljon. Mielestäni kehityin ohjelmoijana yleisesti ja opin etenkin koodin jatkuvan testaamisen tärkeyden. Tämä kävi ilmi, koska virheiden etsintä verkkokoodista on normaalia hankalampaa. Verkkokoodin korjaaminen on erityisen vaikeaa, jos data saapuu perille virheellisenä näennäisen satunnaisesti tai esimerkiksi palvelimella tapahtuva virhe tulee ilmi vasta asiakkaalla.

Myöskin projektin suunnitteluun käytetty aika maksoi itsensä takaisin ohjelmoinnin aikana. Valmiiksi suunniteltu arkkitehtuuri ja suuntaa antava luokkakaavio vähensi huomattavasti virheiden määrää verrattuna aikaisemmin ilman suurempaa suunnittelua kirjoittamaani koodiin. Huolellinen suunnittelu ja nopeat prototyypin omaiset testit myös mahdollistivat suurten, koko arkkitehtuuriin kohdistuvien virheiden huomaamisen jo ennen ohjelmoinnin aloittamista. Tällaisten virheiden korjaaminen olisi saattanut pakottaa minut ohjelmoimaan suuria osia jo valmiista koodista kokonaan uudelleen.

Kokonaisuutena olenki erittäin tyytyväinen opinnäytetyöhöni. Mikä tärkeintä, olen oppinut paljon tulevaisuudessa varmasti hyödyllisiä taitoja sekä tietoa. Olenkin oppinut arvostamaan suunnittelua ja testausta aivan uudella tavalla. Lisäksi olen erittäin positiivisesti yllättynyt opinnäytetyön ohessa syntyneen verkkokirjaston laadusta. Alussa itselleni asettamat laatuvaatimukset olivat huomattavasti matalammalla, kuin mihin lopullinen verkkokirjasto on yltänyt.

LÄHTEET

Allman, M. & Paxson, V. & Blanton, E. 2009: TPC Congestion Control. Pdf-dokumentti. Saatavilla: http://tools.ietf.org/pdf/rfc5681.pdf (Luettu 12.6.2013)

Appleby, A. 2011: MurmurHash3. Web-dokumentti. Saatavilla: http://code.google.com/p/smhasher/wiki/MurmurHash3 (Luettu 10.9.2013).

Armitage, G. & Claypool, M. & Branch, P. 2006: Networking and Online Games: Understanding and Engineering Multiplayer Internet Games. Wiley.

Broekhuizen, J. & Ssekibuule, R. 2004: CORBA in a Real-Time Game Environment. Pdf-dokumentti. Saatavilla: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.7313&rep=rep1&type=pdf (Luettu 19.7.2013).

Clugston, D. 2005: Member Function Pointers and the Fastest Possible C++ Delegates. Web-dokumentti. Saatavilla: http://www.codeproject.com/Articles/7150/Member-Function-Pointers-and-the-Fastest-Possible (Luettu 15.8.2013).

Comer, D. E. 2000: Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (4th Edition). Prentice Hall.

CORBA 2013: CORBA BASICS. Web-dokumentti. Saatavilla: http://www.omg.org/gettingstarted/corbafaq.htm (Luettu 17.7.2013).

Fiedler, G. a. 2008: Reliability and Flow Control. Web-dokumentti. Saatavilla: http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/ (Luettu 2.7.2013).

Fiedler, G. b. 2008: UDP vs. TCP. Web-dokumentti. Saatavilla: http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/ (Luettu 14.6.2013).

Fiedler, G. c. 2008: Virtual Connection over UDP. Web-dokumentti. Saatavilla: http://gafferongames.com/networking-for-game-programmers/virtual-connection-over-udp/ (Luettu 27.6.2013).

Hart, J. M. 2010: Windows System Programming (4th Edition). Addison-Wesley Professional.

Javvin. 2007: Network Protocols Handbook (4th Edition). Javvin Press.

Lengyel, E. 2010: Game Engine Gems: v. 1. Jones and Bartlett Publishers.

Mitchell, M. L. & Samuel, A. & Oldham, J. 2001: Advanced Linux Programming. Kirja saatavilla ilmaiseksi pdf muodossa: http://www.advancedlinuxprogramming.com/alp-folder/advanced-linux-programming.pdf (Ladattu 9.7.2013).

Sawashima, H. & Hori, Y. & Sunahara, H. & Oie, Y. 1997: Characteristics of UDP Packet Loss: Effect of TCP Traffic. Web-dokumentti. Saatavilla: http://www.isoc.org/inet97/proceedings/F3/F3_1.HTM (Luettu 12.6.2013).

Thurlow, R. 2009: RPC: Remote Procedure Call Protocol Specification Version 2. Pdf-dokumentti. Saatavilla: http://tools.ietf.org/pdf/rfc5531.pdf (Luettu 13.7.2013).

Vikram, N. 2005: A Game Engine on Distributed Systems using CORBA. Pdf-dokumentti. Saatavilla: http://vikramnix.tripod.com/PDF_Full-Sem-Final_Report.pdf (Luettu 19.7.2013).